CEWES MSRC/PET TR/98-38

A Fortran 90 Application Programming Interface to the POSIX Threads Library

by

Henry A. Gabb
R. Phillip Bording
S. W. Bova
Clay P. Breshears

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research Corporation

A Fortran 90 Application Programming Interface to the POSIX Threads Library

Henry A. Gabb[φ] R. Phillip Bording[κ] S. W. Bova[λ] Clay P. Breshears[μ]

U.S. Army Corps of Engineers Waterways Experiment Station
Major Shared Resource Center

Abstract

Pthreads is a POSIX standard established to control the spawning, execution, and termination of multiple threads within a single process. Because of a much lower system overhead, use of Pthreads is an attractive approach. Under this programming paradigm on a shared-memory system, threads execute concurrently within a single address space, although multiple processors may be employed to execute the various threads. An obstacle to scientific programming with Pthreads is that no Fortran interface is defined as part of the POSIX standard. We present our current progress in defining and implementing a complete Fortran 90 interface to the Pthreads library. Also presented are many of the design decisions made and lessons learned while striving to keep the bindings as portable as possible. Initial timing results indicate that the per-processor performance may be slightly less than with compiler directives, but that the scalability is superior. In addition, combining Pthreads with MPI under Fortran is discussed. This shows promise to become a useful programming model for clusters of symmetric multiprocessors.

Keywords: POSIX threads, Fortran 90 interface, parallel programming, high-performance computing, SMP clusters.

[φ] CEWES MSRC Computational Migration Group; gabb@ibm.wes.hpc.mil
[κ] CEWES MSRC Computational Migration Group; bording@nrcmail.wes.hpc.mil
[λ] CEWES MSRC On-site CFD Lead for PET; bova@gonzo.wes.hpc.mil
[μ] CEWES MSRC On-site Parallel Tools Lead for PET; clay@turing.wes.hpc.mil

1. Introduction

Pthreads is a POSIX standard established to control the spawning, execution, and termination of multiple tasks within a single process. Concurrent tasks are assigned to independent threads. Threads have local, private memory but also share the memory space of the global process. On symmetric multiprocessors (SMP), the system can run threads in parallel. As useful as the Pthreads is for parallel programming on SMP computers, a Fortran interface is not defined by the standard. However, there are no serious technical barriers to implementing such an API (Application Programming Interface).

Threads yield efficient resource utilization and require less system overhead to maintain. Threads are sometimes called "lightweight" processes since multiple threads created by the same process share the same memory address space. There is no need to save and restore large portions of memory when switching context between threads. This savings of processor time and resources is one of the major advantages of programming with threads. An alternate parallel programming paradigm is provided by compiler directives, which tend to break loops into separate UNIX processes with fork/join operations. These processes are usually linked to physical processors and require a significant amount of overhead to create, maintain, and destroy. When a processor switches context between two processes, the entire memory space of the executing process must be saved and the memory space of the process scheduled for execution must be restored.

One of the most common uses for threads is to overlap computation with I/O. In this case, a process creates one thread to perform I/O and another to continue with calculations. The threads are then executed concurrently. For example, when the I/O thread is waiting for the completion of input, the computation thread executes; when the computation thread waits for a memory fetch, the I/O thread executes.

We describe the implementation of a Fortran 90 API to the Pthreads library developed at the CEWES MSRC. A brief introduction to Pthreads is given in Section 2. Section 3 covers the design decisions made during development as well as the overall structure of the API. An example program illustrating basic usage of the API is also included. A time comparison with other SMP parallel programming models is reported in Section 4. A more complicated program showing how loop-level parallelism can be expressed using Pthreads is shown. The combined Pthreads/MPI programming model is covered in Section 5. Our conclusions are given in Section 6. Hardware issues related to threads programming are discussed in the Appendix.

2. The POSIX Threads Library

2.1 The Pthreads Programming Model

Threads are used for task or function parallelism. A single application consisting of many independent tasks may break up its work into a set of concurrent threads. For example, independent loop iterations (encapsulated within a function call) can be executed as threads. Each thread is created and assigned a given function. The function code is executed concurrently with all other active threads. Upon completing its task, a thread may self-terminate, be cancelled by another thread, or be joined to another thread. Each thread is an instruction stream, with its own stack, sharing a global memory space. Potentially disasterous simultaneous access to global memory is coordinated through mutual exclusion (mutex) variables.

All threads executing within a process are peers. There is no explicit parent-child relationship. A thread may be halted until another thread completes its task, but ultimately, all threaded tasks are concurrent. The operating system performs scheduling and resource allocation. However, Pthreads functions are available to set thread execution priorities and scheduling.

2.2 Library Details

The library is relatively small, consisting of only 61 routines that can loosely be classified into three categories: thread manipulation (e.g., creation, termination), synchronization (e.g., locks, barriers), and scheduling (e.g., execution priority). Each thread is given a unique thread ID upon creation. The Pthreads standard defines attributes in order to control the execution characteristics of threads. Such attributes, which include detach state (whether or not a thread can

be rejoined to another thread upon completing its task), stack address, stack size, scheduling policy, and execution priorities, are usually stored in opaque structures that can only be changed by Pthreads functions. An exhaustive description of Pthreads will not be given here. Several excellent texts dedicated to this subject are available. The book of Nichols et al. [3] was used as a reference for this work.

3. Fortran 90 Interface

3.1 Pthreads Functions

The interface consists of two files. The first is a Fortran 90 module containing some necessary constants and derived type definitions. The second is a collection of wrapper routines that provide the Fortran bindings to the Pthreads library. These wrappers are void C functions that call the vendor-supplied Pthreads library routines. The wrappers are called from Fortran 90 programs as if they were external subroutines.

Many software libraries incorporate both a Fortran and C interface to the same functions regardless of the language in which the libraries are originally implemented. In developing our API, we have taken a cue from some of these previous efforts; for example, [5] and [6]. The wrapper functions are given the same names as the corresponding POSIX routines with an "f" prefixed. For example, a C thread obtains its thread ID with the call

```
my_id = pthread_self();
```

whereas a Fortran thread gets its thread ID as follows

```
call fpthread_self(my_id).
```

Also, most C routines return an error code via the function name. So, an additional parameter is added to the end of the Fortran argument list in order to return this error code.[1]

When designing routines written in one language to be called from code written in another language, matching the methods utilized for the passing of parameters is an important consideration. Fortran 90 uses *pass by reference* while C uses *pass by value*. This difference requires that all C wrapper functions expect an address when called from Fortran; i.e., all dummy arguments are pointers. We were not tempted to mix Fortran 90 pointers with C pointers, although this may have simplified some of the wrapper routines. Reliance on a straightforward compatibility between the two languages would be risky and would likely reduce the portability of the bindings.

Another portability issue is how compiled names are denoted in the object code created by the compiler. The Fortran 90 compiler on the SGI Origin 2000 appends an underscore character to the name of compiled functions. Thus, each of the C wrapper function names is suffixed with an underscore character. Other systems use all capital letters while others do not perform any alterations. To handle this difference between compilers, the wrapper source code contains preprocessor commands that are able to select the appropriate name for functions based on the architecture and resident compiler.

3.2 Pthreads Structures

The Pthreads library makes extensive use of C structures in its definition of Pthreads data types. The Pthreads standard defines data types to handle such things as thread attributes, mutex and conditional variables, and mutex and conditional attributes. To preserve the Pthreads types defined within the bindings, a Fortran 90 derived type is defined with the same naming conventions used to name the wrapper routines; e.g., **`pthread_mutex_t`** becomes **`fpthread_mutex_t`**.

[1] **`pthread_self`** is the only Pthreads routine whose error code is the same as the desired return value. The function **`pthread_testcancel`** returns no error code. Thus, an extra parameter for the error code is not added to these bindings.

Typically, only the functions within the Pthreads library manipulate the data in these structures. The Fortran 90 application program rarely needs to access the data directly. Instead, it must pass the address of a data type to a library routine. Rather than trying to pass Fortran 90 derived types composed of the appropriate components to be interpreted as C structures, the derived types defined in the Fortran 90 module contain only an integer. The wrappers to the POSIX functions interpret this integer as the address of the C structure required. The wrapper code must, where appropriate, allocate space on the heap for the needed structure, decode the integer parameter to be a pointer to a structure, modify the contents of the structure, and free up the space when the structure is no longer needed. Thus, only the structure locations are communicated between the Fortran 90 application and the interface. This improves the portability of the bindings and the application programmer need not worry about any differences between Fortran 90 and C pointers or derived types and structures. Within the Fortran 90 module, this integer is often declared to be **`PRIVATE`**. This adds an extra level of security to these derived types by preventing the programmer from inadvertently changing the address of an important variable. (Note: The one execption to this scheme is **`fpthread_t`** which is a derived type containing a single integer, but this integer is **`PUBLIC`** and holds the integer thread id in the component **`thread`**.)

An example of this information hiding is illustrated in the following two wrapper codes. The first is **`fpthread_mutex_init_`** which allocates memory to hold the mutex structure (**`pthread_mutex_t`**), initializes the structure by calling the Pthreads function, and returns the address (as an integer) of the initialized mutex to the calling Fortran 90 code.

```
void fpthread_mutex_init_(int *mutex, int *attr, int *ierr)
{
      pthread_mutex_t *lmutex;
      pthread_mutexattr_t *lattr;

      lmutex = (pthread_mutex_t *) malloc( sizeof (pthread_mutex_t));
      lattr = (pthread_mutexattr_t *) (*attr);
      *ierr = pthread_mutex_init(lmutex, lattr);
      *mutex = (int) (lmutex);
}
```

The second wrapper is **`fpthread_mutex_destroy_`** which returns the memory allocated to the mutex back to the system and returns the NULL pointer to the calling Fortran 90 code.

```
void fpthread_mutex_destroy_(int *mutex, int *ierr)
{
      pthread_mutex_t *lmutex;

      lmutex = (pthread_mutex_t *) (*mutex);
      *ierr = pthread_mutex_destroy(lmutex);
      free(lmutex);
      *mutex = NULL;
}
```

The programmer must pay close attention to the scope of variables since the rules for scoping are different in Fortran and C. This is particularly important when using mutexes and conditional variables. In order to ensure that only a single thread is allowed to enter a critical section of the code, a single copy of a lock must be declared in memory and visible to all threads that make use of it. Such global variables are easily declared in C. With Fortran 90, it is suggested that all mutex locks and conditional variables be declared within a module that is used by each subroutine in which these synchronization constructs are needed.

3.3 Example Program

In this section we present an example illustrating the use of the API. This trivial application creates four threads that identify themselves and are rejoined to the parent program. For brevity, comments and error checking are excluded.

```
program hello
   use fpthread
   implicit none

   integer                          :: i, ierr
   integer, parameter               :: N = 4
   type (fpthread_t), dimension(N) :: tid
   integer, dimension(N)            :: exitcodes
   type (fpthread_mutex_t)          :: print_mutex

   common print_mutex
   external fpthread_mutex_init, fpthread_create, fpthread_join

   call fpthread_mutex_init(print_mutex, NULL, ierr)
   do i = 1, N
      call fpthread_create(tid(i), NULL, say_hello, NULL, ierr)
   enddo
   do i = 1, N
      call fpthread_join(tid(i), exitcodes(i), ierr)
   enddo
end program hello

subroutine say_hello
   use fpthread
   implicit none

   type (fpthread_t)       :: my_id
   type (fpthread_mutex_t) :: print_mutex
   integer                 :: ierr

   common print_mutex
   external fpthread_self, fpthread_mutex_lock, fpthread_mutex_unlock

   call fpthread_self( my_id )
   call fpthread_mutex_lock(print_mutex, ierr)
      print*, 'Hello from thread ', my_id%thread
   call fpthread_mutex_unlock(print_mutex, ierr)
end subroutine say_hello
```

NULL arguments instruct Pthreads to use the system defaults. Since Fortran 90 has no concept of **NULL**, an integer variable called **NULL** is declared and initialized to zero. The Fortran 90 module, **fpthread**, contains the type definitions and constants described by the Pthreads standard, as well as **NULL**. A mutex variable, **print_mutex**, is declared and initialized with a call to **fpthread_mutex_init**. The mutex variable is placed in **COMMON**[2] so that all threads can check whether it is locked or unlocked. Without this mutex, the threaded output could become scrambled and unreadable. Four separate threads for the subroutine **say_hello** are created with calls to **fpthread_create**. Unique thread identification numbers are placed in the array, **tid**. Pthreads can self-terminate by calling **fpthread_exit**. In this example, however, the main program would likely terminate before all threads have the opportunity to identify themselves. The **fpthread_join** loop acts as a barrier. The main program cannot continue until all threads are joined.

[2] Modules are preferred in modern Fortran.

4. Preliminary Results

To test the efficiency of threaded computation, a C/Pthreads program[3] to compute π was translated to Fortran 90 using our API. The following program illustrates several features of Pthreads programming. Once again, comments and error checking are excluded for brevity. The program performs a reduction operation so a mutex variable is required to synchronize access to the accumulated sum. All threads must have access to this mutex, so it is placed in a module. In fact, all global variables are declared in this module.

```
module pi_module
   use fpthread
   implicit none

   type (fpthread_mutex_t)                      :: reduction_mutex
   type (fpthread_t), dimension(:), allocatable :: tid

   integer :: intervals, num_threads
   real    :: pi, n_1
end module pi_module


program compute_pi
   use fpthread
   use pi_module
   implicit none

   integer :: i, ierr

   external PIworker
   external fpthread_mutex_init, fpthread_create, fpthread_join

   print*, 'How many intervals and threads?'
   read*, intervals, num_threads
   allocate( tid(num_threads) )

   n_1 = 1.0 / real( intervals )
   pi = 0.0

   call fpthread_mutex_init(reduction_mutex, NULL, ierr)

   do i = 1, num_threads
      call fpthread_create(tid(i), NULL, PI_worker, NULL, ierr)
   enddo

   do i = 1, num_threads
      call fpthread_join(tid(i), NULL, ierr)
   enddo
   deallocate( tid )

   print*, 'Computed pi = ', pi
end program compute_pi
```

[3] Taken from an example program in Leo Dagum and Ramesh Menon, "OpenMP: A Proposed Industry Standard API for Shared Memory Programming," IEEE Computational Science and Engineering, Vol. 5, pp. 46-55, 1998.

```
subroutine PI_worker
   use fpthread
   use pi_module
   implicit none

   type (fpthread_t) :: my_num
   integer           :: i, my_id, ierr
   real              :: sum, my_pi, x, f

   external f
   external fpthread_self, fpthread_mutex_lock, fpthread_mutex_unlock

   call fpthread_self( my_num )
   my_id = my_num%thread – minval( tid(:)%thread ) + 1
   sum = 0.0

   do i = myid, intervals, num_threads
      x = n_1 * ( real(i) - 0.5 )
      sum = sum + f(x)
   enddo
   my_pi = n_1 * sum

   call fpthread_mutex_lock(reduction_mutex, ierr)
      pi = pi + my_pi
   call fpthread_mutex_unlock(reduction_mutex, ierr)
end subroutine PI_worker


function f(a)
   real :: f, a
   f = 4.0 / (1.0 + a * a)
end function f
```

This program shows how loop-level parallelism can be expressed using Pthreads. The loop counter in **`PI_worker`** is determined at run-time using thread id numbers for the starting index while the increment is set to the number of threads. The programmer controls the grain size of the computations through the number of threads created. More threads give finer grained parallelism and vice versa.

We compare the parallel performance on a 16-processor SGI Origin2000 of the iterative π computation using Fortran with Parallel Computing Forum (PCF) directives, C with preprocessor pragmas, C/Pthreads, and Fortran/Pthreads.[4] Pragmas and PCF directives create multiple UNIX processes which are linked to physical processors. Therefore, the number of threads cannot exceed the number of processors.

Table 1 shows that there is insufficient work to merit parallel execution. Notice that the PCF and pragma programs slow down as more processors are added. Resource allocation is left to the operating system for Pthreads programs. The operating system does not attempt to parallelize such a small amount of work so performance does not suffer. When the number of intervals is increased by an order-of-magnitude (Table 2), efficient parallelism is more important. Once again, the operating system does not parallelize the Pthreads programs until the number of threads exceeds eight. This experiment indicates that threads give better scalability than preprocessor or compiler directives. Tables 2 and 3 show that resource utilization and overall speedup is better for the Pthreads programs. We acknowledge at this point, however, that the Pthreads programs are more complicated. Pthreads, after all, is a low-level library.

[4] The test programs are available from the authors.

# of threads	1	2	4	8	16
PCF directives (F90)	0.11	0.35	0.24	0.19	0.47
Pragmas (C)	0.11	0.23	0.20	0.21	0.24
Pthreads (C)	0.11	0.11	0.11	0.11	0.09
Pthreads (F90)	0.17	0.17	0.17	0.17	0.13

Table 1. Time comparisons on a 16-processor SGI Origin2000 for π computations using different parallelization techniques. Number of intervals equals 10^6. Time is in seconds from the function **`gettimeofday()`**.

# of threads	1	2	4	8	16	64	256	1000
PCF directives (F90)	1.09	1.10	0.84	0.53	0.67	--	--	--
Pragmas (C)	1.09	0.77	0.59	0.40	0.40	--	--	--
Pthreads (C)	1.09	1.09	1.09	1.07	0.62	0.18	0.24	0.35
Pthreads (F90)	1.76	1.76	1.76	1.57	0.84	0.25	0.23	0.29

Table 2. Time comparisons on a 16-processor SGI Origin2000 for π computations using different parallelization techniques. Number of intervals equals 10^7. Time is in seconds from the function **`gettimeofday()`**.

# of threads	1	2	4	8	16	64	256	1000
PCF directives (F90)	10.88	11.66	6.25	3.35	2.12	--	--	--
Pragmas (C)	10.85	5.94	3.20	1.74	1.10	--	--	--
Pthreads (C)	10.95	10.47	6.27	3.61	2.06	1.00	0.68	0.52
Pthreads (F90)	17.59	13.78	7.95	4.48	2.46	1.11	0.63	0.68

Table 3. Time comparisons on a 16-processor SGI Origin2000 for π computations using different parallelization techniques. Number of intervals equals 10^8. Time is in seconds from the function **`gettimeofday()`**.

5. Pthreads and MPI

A current trend in the future of HPC architectures is the cluster of SMP nodes connected to each other via a network. Each SMP node is a shared memory multi-processor that can make use of Pthreads. In order to share data between nodes, some explicit message passing must be done. Thus, a combination of both Pthreads (local to SMP nodes) and MPI (message passing between nodes) is a potential model of computation for effectively programming on SMP cluster platforms.

In order to test the efficacy of such a hybrid model, we have developed small test codes that use the Fortran 90 Pthreads API and MPI calls. Our experiments were carried out on the SGI/CRAY T3E at CEWES MSRC. Each node of the T3E simulated a SMP cluster running multiple threads while data was shared between nodes via MPI calls. Each thread was able to communicate directly with other threads on other processors. The rank of the processor was used to address messages toward a pool of threads executing on the processor. Tag numbers were then used by individual threads to receive messages that were specifically addressed to that thread.

The ASCI Project sPPM benchmark [4] combines Pthreads with MPI. However, a single thread on each processor is chosen to be the message-passing liason between all other thread groups. Haines et al. [2] have described Chant, an extension to the POSIX threads library with functions that facilitate communication and synchronization among threads executing within a distributed-memory environment. The Chant package was designed to work with MPI and Pthreads and tested on an Intel Paragon and network of SUN workstations. It is unclear whether further development has been carried out on the Chant library.

6. Conclusions

Pthreads is a convenient method of expressing task-level parallelism. The programmer has more control over scheduling and synchronization with Pthreads than with parallel compiler directives. More important, the programmer can control the grain size of each thread. If the grain is too coarse, more threads can be created and vice versa. Pthreads has the advantage of low system overhead because all threads exist in a single UNIX process. In addition, multiple threads can exist on a single processor, thus giving better resource utilization.

Threading is widely used in systems programming, where C/C++ is the primary language. However, Fortran is the primary language for high performance computing (HPC). Threading is not as common in HPC applications because the Posix standard does not define a Fortran interface to the Pthreads library. The Fortran API described here provides an additional parallel programming tool to the HPC community. In addition, combining Pthreads with MPI presents an attractive programming model for SMP clusters.

Acknowledgements

This work was funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through Programming Environment and Training (PET), Contract Number: DAHC 94-96-C0002, Nichols Research Corporation.

References

Alfieri, R.A., “An Efficient Kernel-Based Implementation of POSIX Threads”, USENIX, June 1994.

Haines, M., Cronk, D. and Mehrotra, P., “An Introduction to Chant,” *Bulletin* of the Technical Committee on Operating Systems and Applications Environments, IEEE Computer Society, Spring 1994.

Nichols, B., Buttlar, D. and Farrell, J., *Pthreads Programming*, O’Reilly and Associates, Inc., Sebastopol, CA, 1996.

Owens, J., “The ASCI sPPM Benchmark Code,” http://www.llnl.gov/asci_benchmarks/asci/limited/ppm/, 1996.

Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W. and Dongarra, J., *MPI: The Complete Reference*, The MIT Press, Cambridge, Massachusetts, 1997.

Sunderam, V. S., Geist, G. A., Dongarra, J. and Manchek, R., “The PVM Concurrent Computing System: Evolution, Experiences, and Trends,” *Parallel Computing*, Vol. 20, No. 4, pp.531-545, April 1994.

Appendix. Threads and Hardware

Historically, program instruction execution sequences are based on incrementally advancing a hardware program counter until a logic control action requires redirection. These program jumps result from logical tests, change in machine state, or error states detected during instruction execution. These program counter and program jump features are essential to von Neumann machines. A computing engine must consist of at least one program counter. One source of program state changes are the input/output (I/O) functions. Many I/O actions can require significant time delays during execution, and by changing program state this idle time could be used productively. The system can either wait until the I/O task indicates completion or the program counter can be redirected to execute other code. The latency of I/O actions can be measured in terms of hundreds to thousands of machine execution cycles.

This redirection to another task required changes to the operating system to maintain a queue of tasks. Each task is executed based on a notion of priority or importance. As these queue-based changes evolved it soon became apparent that some I/O operations could be executed independently on simple dedicated processors. These extra computer execution units were called "direct memory access" (DMA) devices or I/O channels.

Once these ideas were in hardware and supported by software they resulted in significant machine performance improvements. The natural extension of these support or peripheral processors is to make them more powerful and more general. Another approach is to make all the computing elements look similar and multi-functional.

Threads are the modern realization of the task queue. The operating system processes and user programs consist of many threads, and as a program executes it switches between the many threads of the total program thread set. The switching time impacts the total time available for computing. Again the context switch latency is measured in machine cycles, and for example, 200 cycles are required on the MC88110 [1]. To change from task to task can be a software process or can be supported by hardware. Any von Neumann style machine can support threads in software. Special hardware can also support threads by enabling context switching. The limit of latency would be for each instruction to execute from a different thread. If a thread completed its task, then a new thread would be selected from the queue. The hardware would simply point to a different program counter and register set when the context is switched. This has a novel feature of hiding the DRAM memory latency problem. A 60 nanosecond access time memory and a 400 megahertz clock has a latency of 24 instruction cycles. If the hardware supports enough threads to hide memory access times then processors will not suffer the wait state loss now being experienced. Further, more thread contexts could exist than processors and from a single memory environment each would execute based on some notion of priority. Some of these ideas come from the new thread based hardware machines being built and tested by Tera Computer Company.